UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

TOOLS AND TECHNIQUES FOR THE STATIC VERIFICATION OF PROGRESS IN COMMUNICATION-CENTRED SYSTEMS

André Filipe Marinhas Henriques da Silva Camacho

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software
2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

TOOLS AND TECHNIQUES FOR THE STATIC VERIFICATION OF PROGRESS IN COMMUNICATION-CENTRED SYSTEMS

André Filipe Marinhas Henriques da Silva Camacho

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada por: Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos e Prof. Doutor Hugo Filipe Mendes Torres Vieira

2014

Agradecimentos

Durante este último ano, que terminou com o desenvolvimento deste trabalho, recebi o apoio de diversas pessoas. Quero começar por agradecer aos meus orientadores, Prof. Vasco Vasconcelos e Prof. Hugo Vieira pela disponibilidade, apoio constante e também pela oportunidade de realizar um projeto nesta área.

Quero também agradecer ao Departamento de Informática da FCUL e a todos os meus colegas que ao longo destes últimos anos, ajudaram a alcançar todos os meus objectivos. Um especial obrigado a todos os meus colegas do LaSIGE pela constante disponibilidade para ajudar e pelo fantástico ambiente de trabalho e estudo proporcionado.

Por último, mas não menos importantes, um grande obrigado à minha família e amigos pelo apoio incondicional que me deram em todos os momentos ao longo do meu caminho. Em especial aos meus pais Maria do Carmo e João Manuel, agradeço todos os momentos de apoio e incentivo, que foram fundamentais. Obrigado por terem acreditado sempre em mim, pois sem o vosso apoio não teria sido possível alcançar esta meta.

Aos meus pais.

Resumo

Com a crescente disseminação da computação distribuída nos últimos anos, a forma como prevenimos os erros de programas concorrentes é cada vez mais importante. O desenvolvimento de programas concorrentes é difícil, exigindo ao programador um grande esforço para tentar "prever" possíveis estados futuros do programa. Num sistema concorrente a execução pode ter um amplo número de caminhos possíveis cujo resultado pode ser inesperado. Mesmo realizando intensivos testes de software, pode-se nunca chegar a explorar a sequência necessária para encontrar os caminhos que levam a certos erros, tornando estes testes insuficientes. Um dos erros difíceis de detetar é a ocorrência de pontos de bloqueio, designados por *deadlocks*.

Um sistema possui a propriedade de *progresso*, quando na sua execução não ocorrem pontos de bloqueio. É necessário provar que nenhum caminho possível revela surpresas, para garantir que existe progresso no sistema. A nossa abordagem procura garantir a ausência de pontos de bloqueio, para além de outras propriedades de correcção, implementando técnicas de verificação a serem usadas em ferramentas de análise estática (apenas inspecionando o código fonte). A verificação vai ser realizada em sistemas concorrentes baseados em troca de mensagens, onde vão ser garantidas as propriedades de *fidelidade* e progresso. Garantindo estas propriedades é possível ter a certeza que as mensagens trocadas seguem um protocolo bem definido e que o sistema está livre de pontos de bloqueio.

A linguagem de modelação de sistemas usada no contexto deste trabalho é baseada no calculo π introduzido por Milner, Parrow e Walker [5, 6], um modelo universal de computação que permite modelar e especificar de uma forma precisa sistemas de processos concorrentes, servindo posteriormente de suporte para técnicas rigorosas de análise de propriedades.

Para garantir a propriedade de progresso vai ser usado um sistema de tipos que unifica a noção de evento com os tipos de sessão introduzido por Honda, Kubo e Vasconcelos [2, 3] que permitem descrever o protocolo de comunicação entre dois participantes ponto-a-ponto. O uso da noção de evento introduzido por Vieira e Vasconcelos [9], permite capturar as dependências das comunicações entre processos e suporta a verificação que estas estão bem estruturadas.

Como objetivo, vão ser endereçadas comunicações *ponto-a-ponto* sendo-nos dadas as especificações dos tipos (incluindo as anotações de eventos) e a ordenação de eventos, por forma a conferir se o programa está de acordo com as especificações.

Palavras-chave: Verificação de Software, Sistema de tipos, Fidelidade, Progresso, Concorrência.

Abstract

With the growing dissemination of distributed computing in recent years, preventing runtime errors in concurrent programs is increasingly important. The development of concurrent programs is difficult, requiring from the programmer a great effort to "predict" possible future states of the program.

In a concurrent system the execution can have a large number of possible paths, some of which leading to unexpected behaviours. Even performing intensive software testing, the explored sequence may never find the paths that lead to certain errors, making such testing incomplete. *Deadlocks* are one of the hard to detect errors in concurrent systems. We say that a system has the property of *progress* when its execution does not lead to deadlocks. It is necessary to prove that no path reveals any surprises to ensure that there is progress in the system. Our approach aims to ensure the absence of deadlocks in addition to other correctness properties, implementing verification techniques to be used in static analysis tools (that just inspect the source code). The analysis will be performed on concurrent systems based on message exchanges, targeting the properties of *fidelity* and progress. Ensuring such properties means guaranteeing that the message exchanges follow a well-defined protocol and that the system is free from deadlocks.

The language used to model systems in the context of this work is based on the π-Calculus introduced by Milner, Parrow and Walker [5, 6], a universal model of computation which allows to model and specify concurrent systems in a precise way, serving subsequently as a support for rigorous analysis techniques. To ensure progress, we will use a type system that unifies a notion of event with the session types introduced by Honda, Kubo and Vasconcelos [2, 3] which allow to describe the communication protocol between two participants (point-to-point). The use of events introduce by Vieira and Vasconcelos [9], allows to capture the communication dependencies between processes so as to support the verification that such dependencies are well structured.

Our goal will be to support *point-to-point* communications, considering the type specifications (including the event annotations) and the orderings of events are given, which allow us to check if the program is in accordance with the specifications.

Keywords: Software Verification, Type System, Fidelity, Progress, Concurrency.

Contents

Chapter 1

Introduction

1.1 Motivation

Concurrency in computational systems is increasing with the expansion of distributed infrastructures of computation. Distributed software applications and multi-core architectures involve a large amount of communications. It is therefore essential to guarantee correct interactions to have a reliable system that does what it is supposed to. Two fundamental correctness properties are fidelity and progress.

We say a system has (protocol) fidelity when the communication between the participants follows the prescribed protocols. For example when a process is waiting for a message containing a string and instead receives an integer, we say that there is no fidelity because of the exchanged value type. Another simple example is when a process is ready to output a message but there is no other process to receive the message as in $p_1!x$, so that the process does not follow the introduced protocol.

We say a system has progress when it never incurs in deadlocks. A deadlock occurs when a process A is holding on to some resource that a process B needs, while at the same time, the process B is holding on to a different resource that process A needs. In other words, when there is a cyclic dependency involving the processes and the allocated resources.

The main motivation of this work is to verify the absence of deadlocks in addition to other correctness properties. To accomplish this goal we implement verification techniques to be used in static analysis tools. By detecting deadlocks previous to potential execution the programmer prevents the deployment of programs with deadlocks, which subsequently obviate the need to deadlock debugging, a difficult to carry out and very time consuming task. In particular, it is difficult to get the same deadlock error intentionally when debugging in a system with a extensive number of possible paths.

1.2 Goals

The main goal of this work is the design and development of a static analysis tool that guarantees fidelity and progress for message passing concurrent systems.

To model the concurrent systems in the context of this work we use a language that is based on the π-Calculus introduced by Milner, Parrow and Walker [6], which allows to model and specify concurrent systems in a rigorous way, and subsequently supports exact analysis techniques which allows us to assure the properties in all the possible "paths" of a concurrent system.

To achieve the proposed work, we develop an analysis tool with support to point-to-point communication. Namely we use a type system that unifies session types introduced by Honda, Kubo and Vasconcelos [2, 3] with a notion of *event* following the approach presented by Vieira and Vasconcelos [9]. Such event information allows to capture the communication dependencies between processes and gives support to the verification that such dependencies are well structured. The necessary type specifications (with event annotations) and the orderings of events are provided by the programmer. Our analysis tool statically analyses the source code and guarantees that message exchanges follow a given protocol and that the system is free from deadlocks.

1.3 Contributions

The main contributions of our work are:

- Implementation of a type checking procedure that unifies the notion of session types with events which involved refining the theoretical type system in which our work is based.

- Development of a static verification tool which ensures progress in point-to-point communications, that may be used via command line or via an Eclipse plugin.

1.4 Structure of the document

The next chapters of this thesis are structured as follows:

Chapter 2
Describes the background work and introduces necessary concepts, including the pi-calculus, session types and presents the notion of event fundamental to our work.

Chapter 3
Introduces our static verification tool and describes how the tool works, explaining each of the elements of the input specification that the tools needs the user to specify. Also, we present an account of the kind of errors that can be signalled by the tool, and we present a series of illustrative examples.

Chapter 4

Description of the implementation of the tool, technologies employed, including a detailed account of the programming elements involved and a description of the main steps of the verification program.

Chapter 5

Presents the conclusions and future work.

Chapter 2

Background

This chapter introduces essential concepts necessary to our development. We first introduce the π-calculus [5, 6], a universal model of concurrent communicating processes that will be used as our modelling language for concurrent systems. Then we introduce *session types* [3, 2] which allow to describe the interaction between different processes in a concurrent system, and may be used to ensure *protocol fidelity* in the communication between the processes. Finally we introduce *session types with events* [9] which add to session types an abstract notion of moment in time by adding an event annotation, which will be used in our approach to assure progress of systems. We focus our presentation on examples that help in providing some intuition about the main notions involved in our work.

2.1 The π-calculus

The π-calculus [5, 6] is a foundational model of concurrent communicating processes, which allows to focus on the communication of concurrent processes where interaction is based on sending and receiving on channels. Channel identifiers grant access to the communication channel itself. For instance, if x is a channel identifier then $x!$ represents an output on channel x and $x?$ represents an input on channel x. In the input communications it is necessary to specify the bound variable to be instantiated by the received value; for example the input $x?y$ specifies a reception from channel x of a value which instantiates y in the continuation process. Accordingly in the output communications it is necessary to specify the value to be sent on the channel; for example the output $z!$"`hello`" specifies the output of string "`hello`" on channel z. A fundamental notion of this model is that channel identifiers may be passed around in communications, allowing for processes to gain access to channels previously unknown to them.

The example below illustrates a simple scenario where three partners interact. The parallel composition (denoted by $|$) used in the specification of process $System$ describes

three processes which are simultaneously (concurrently) active.

$$\begin{array}{rcl} Bob & \triangleq & (vchat)\, friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\ Alice & \triangleq & friend?y.y?q.y!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.y?answer \\ Carol & \triangleq & oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \\ System & \triangleq & Bob \mid Alice \mid Carol \end{array} \tag{2.1}$$

The Bob process creates a new name $chat$ and sends it on channel $friend$ to $Alice$ that receives the identifier $chat$ on the input $friend?y$. This allows Bob and $Alice$ to share a private channel ($chat$) after the synchronization on friend:

$$\begin{array}{ll} (vchat) & (chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\ & \mid chat?q.chat!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.chat?answer) \\ & \mid oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \end{array}$$

Bob and $Alice$ may now privately (without external interference) interact in channel $chat$, as its identity is known only to them, leading the first of such interactions to the system:

$$\begin{array}{l} (vchat)(chat?a.oldfriend!chat \\ \mid chat!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.chat?answer) \\ \mid oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \end{array}$$

Process Bob sent greetings to process $Alice$ on channel $chat$. At this point, a second interaction in channel $chat$ can take place, leading the system to the following configuration:

$$\begin{array}{l} (vchat)(oldfriend!chat \\ \mid chat?answer) \\ \mid oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \end{array}$$

The process $Alice$ sent a text message to process Bob, sending greetings and asking who is his friend via channel $chat$. At this point, process Bob is willing to share the identity of channel $chat$ with process $Carol$:

$$(vchat)\,(\mathbf{0} \mid (\, chat?answer \mid chat!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”}))$$

After sending channel $chat$ to process $Carol$ on channel $oldfriend$ process Bob has no other actions. At this point, process $Carol$ can use channel $chat$ to communicate with $Alice$ directly, sending “`Hi Alice, my name is Carol.`” and leading the system to the inactive state below, which is structural congruent to the terminated process $\mathbf{0}$.

$$(vchat)\,(\mathbf{0} \mid \mathbf{0} \mid \mathbf{0})$$

2.1.1 π-calculus language specification

Figure 2.1 shows the syntax of the language. The inactive process is denoted by $\mathbf{0}$. The output communication is denoted by $x!y.P$ which represents a process that sends y on the channel x and then proceeds as process P. Symmetrically, the input communication is represented by $x?y.P$ which represents a process that is waiting to input on channel x a

$P ::=$	Processes
$\mathbf{0}$	Inaction
$x!y.P$	Output
$x?y.P$	Input
$*x?y.P$	Replicated input
$P \mid Q$	Paralel composition
$(vx)P$	Name restricion

Figure 2.1: Syntax of processes

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$
$$P \equiv_\alpha Q \Rightarrow P \equiv Q \qquad (vx)\mathbf{0} \equiv \mathbf{0} \qquad (vx)(vy)\,P \equiv (vy)(vx)\,P$$
$$P_1 \mid (vx)P_2 \equiv (vx)(P_1|P_2) \quad (x \notin fn(P_1))$$

Figure 2.2: Structural Congruence

value which instantiates the bound variable z in the continuation process P. The replicated input is similar to the input with the difference that the process survives in fraction to continuously receive a name from the channel and proceed as process P. The parallel composition $P|Q$ represents two processes simultaneously active. The name restriction $(vx)P$ represents a process P with a local (private) name x. The occurrences of x in (vx) and of y in $x?y$ and in $*x?y$ are binding with scope P in processes $(vx)\,P$, $x?y.P$ and $*x?y.P$, respectively. We represent by $fn(P)$ the set of free names of P, defined as expected, for the definition of bound names.

We next define the operational semantics of the language based on the notions of structural equivalence and of reduction. First we introduce the definition of α-equivalence and Structural Congruence

Definition 2.1.1. *(α-equivalence) We say that two processes P and Q are alpha-equivalent, denoted by $P \equiv_\alpha Q$ if P and Q are equal up to a renaming of their bound names.*

Definition 2.1.2. *(Structural Congruence). We say that two processes P and Q are structurally congruent $P \equiv Q$, if we can transform one in the other by using the least congruence relation over processes that satisfies the equations in Figure 2.2.*

The rules of Structural Congruence are given in Figure 2.2. The first rule states that the terminated process $\mathbf{0}$ is the neutral element in the parallel composition. The other two rules in the first line state that parallel composition is commutative and associative. On the second line, the first rule specifies the equivalence between two process that are alpha equivalent, the second gives the possibility to elide or introduce an unused restriction and the third rule shows the equivalence when the order of the bindings is changed. The last rule, which is called scope extrusion, allows the scope of a binder to be extended to a

$$\frac{}{x?y.P \mid x!z.Q \;\rightarrow\; P[z/y] \mid Q} \qquad \frac{P \rightarrow Q}{(vx)P \rightarrow (vx)Q} \qquad \text{(R-Com , R-New)}$$

$$\frac{}{*x?y.P \mid x!z.Q \;\rightarrow\; *x?y.P \mid P[z/y] \mid Q} \qquad \frac{P \;\rightarrow\; P'}{P \mid Q \;\rightarrow\; P' \mid Q} \qquad \text{(R-Rep , R-Par)}$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \;\rightarrow\; Q} \qquad \text{(R-Cong)}$$

Figure 2.3: Reduction rules

process which does not have free occurrences of the bound name. We may now introduce the Reduction relation which explains how systems evolve.

Definition 2.1.3. *(Reduction) The reduction relation between processes is given by the least relation that satisfies the rules in Figure 2.3 .*

Figure 2.3 presents the reduction relation rules. Rule (R-Com) captures the synchronization between an input and an output: the process $x?y.P$ receives a name that is sent by $x!z.P$ — notice that z replaces y in the continuation process P. Rule (R-Rep) is similar to (R-Com) with the difference that the replicated input is again in the resulting state ready for further synchronizations. Rule (R-New) allows reduction to happen underneath scope restriction. Rule (R-Par) allows reduction to happen in a part of a parallel composition. The remaining rule (R-Cong) introduces structural congruence in the reduction relation, stating that processes that are structurally equivalent have the same behaviour.

For the sake of illustration, we go back to Bob, $Carol$ and $Alice$ conversation, introduced in Example 2.1 reproduced below focusing on the first reduction step.

$$\begin{array}{rcl} Bob & \triangleq & (vchat)\, friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\ Alice & \triangleq & friend?y.y?q.y!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.y?answer \\ Carol & \triangleq & oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \\ System & \triangleq & Bob \mid Alice \mid Carol \end{array} \tag{2.1}$$

The example above shows the initial state of the system, which after a first reduction step is in the state bellow.

$$\begin{array}{rl} (vchat) & (chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\ & \mid\ chat?q.chat!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.chat?answer) \\ & \mid\ oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \end{array}$$

We illustrate a derivation using the Example 2.1, in particular regarding the synchronization on channel $friend$ between Bob and $Alice$. The first rule applied is the reduction rule (R-Cong) (see Figure 2.3). In the context of this rule the *scope extrusion* rule is applied, which allows the scope to be extended to $Alice$. The enlargement of the scope will allow name $chat$ to be sent to $Alice$ underneath the scope of the name restriction.

$$(vchat)friend!chat.R \mid friend?y.S \mid Carol$$
$$\equiv$$
$$(vchat)(friend?y.S \mid friend!chat.R) \mid Carol$$

The second rule applied is (R-Par) that grants the reduction between Bob and $Alice$ while $Carol$ remains unchanged.

$$\frac{(vchat)(friend?y.S \mid friend!chat.R) \;\rightarrow\; P}{(vchat)(friend?y.S \mid friend!chat.R) \mid Carol \;\rightarrow\; P \mid Carol}$$

The third rule applied is (R-New) which allows reduction to happen underneath scope restriction, where P is $(vchat)\,Q$.

$$\frac{friend?y.S \mid friend!chat.R \;\rightarrow\; Q}{(vchat)\,(friend?y.S \mid friend!chat.R) \;\rightarrow\; (vchat)\,Q}$$

Finally the rule (R-Com) is applied so the synchronization takes place on the channel $friend$ between $Alice$ and Bob allowing Bob to send name $chat$ to $Alice$, where R is $y?q.y!$“`Hello Bob, who is your friend ?`”$.y?answer$ and S is $chat!$“`Hello Alice!`”. $chat?a.oldfriend!chat$

$$\overline{(friend?y.R \mid friend!chat.S) \;\rightarrow\; R[chat/y] \mid S}$$

Having presented the process specification language we next turn to presenting session types that can be used to discipline interaction in message passing programs.

2.2 Session types

Session types [3, 2] allow to describe the interaction between different processes in a concurrent system. One of the main goals of session types is to avoid runtime errors via static analysis of the code, guaranteeing systems enjoy the *protocol fidelity* property, which guarantees a well-established communication without unexpected communicating actions from the participants. Without a well-established protocol there is no guarantee that the system will do what is supposed to. This is possible to establish by describing the protocol that represents the interactions among the participants in the communication, establishing for each participant its role in the interaction. Once the protocol is established, the type checking procedure assures the absence of communication errors at runtime.

To understand this concept, we need to imagine a communication between two processes on a channel, each one accessing one end point of the channel. Taking as example a variable x which by itself gives access to any of the two end points of the channel, if x is going to be used to send a boolean and then an integer the session type that describes such usage of channel x is !boolean.!integer.end. The dual session type, which describes the other endpoint in the communication is ?boolean.?integer.end. The ! represents the sending and ? the receiving of a message followed by the type of the value carried in

the message. The end type means that no further interaction is going to happen in the channel.

An important concept is *communication safety*. For instance, if some process is waiting for a boolean value and the other process sends a string value, it may lead to a runtime error. This difference between the type of the value sent and the type of the expected value is a problem for the receiving process which will use the value as if it is of the expected type, leading to possible further errors in the system. This kind of errors are excluded by communication safety (and also by fidelity which entails communication safety).

Vasconcelos [8] introduces a reconstruction of session types in a linear π-calculus where types are also qualified as linear or unrestricted. A linear channel must occur in just two threads, without interference from other threads, and in opposition an unrestricted channel may appear in a unbounded number of threads, thus being shared.

Using a type of *handle* annotation this can be specified accordingly. For example, the un ? (lin ? Integer) adds to the description that the channel can be used zero or more times, in zero or more threads while the received channel (lin ? Integer) must be used in exactly one thread (once to receive an integer), because it is linear.

The following four examples, illustrate four simple similar cases of session types usage, with different situations. We use the exact same example as before (Example 2.1), represented bellow to explain session types. Bob sends a string to $Alice$, then receives her question and finally sends the communication channel to process $Carol$ that replies directly to process $Alice$. This is how the communication should flow between the three processes.

$$\begin{aligned} Bob &\triangleq (vchat)\, friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\ Alice &\triangleq friend?y.y?q.y!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.y?answer \\ Carol &\triangleq oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \\ System &\triangleq Bob \mid Alice \mid Carol \end{aligned} \tag{2.1}$$

The different points of view of channel uses, from each participant are defined by :

$$friend : \mathsf{lin}\,?\,(\,\mathsf{lin}\,?\,\mathsf{string}.\,\mathsf{lin}\,!\,\mathsf{string}.\,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end} \vdash Alice$$

From the point of view of $Alice$, the channel $friend$ is going to be used to receive a linear channel. The received channel is used to receive a string, send a string and receive a string in this order.

$$oldfriend : \mathsf{lin}\,?\,(\,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end} \vdash Carol$$

From the point of view of $Carol$, the channel $oldfriend$ is going to be used to receive a linear channel. The received channel is used to send a string.

$$\begin{aligned} friend &: \mathsf{lin}\,!\,(\,\mathsf{lin}\,?\,\mathsf{string}.\,\mathsf{lin}\,!\,\mathsf{string}.\,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end}, \\ oldfriend &: \mathsf{lin}\,!\,(\,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end} \vdash Bob \end{aligned}$$

From the point of view of Bob, the channel $friend$ is going to be used to send a linear channel that is used to receive a string, send a string and receive a string. The channel $oldfriend$ is used to send a channel that in its turn is used to send a string. Notice the types of the usages delegated by Bob exactly match the usages performed by $Alice$ and

$Carol$, as in fact it is Bob that distributes the $chat$ channel.

Analysing the code of the example we can statically guarantee that we have communication safety and fidelity. We now introduce same variants of the system where there is no fidelity.

$$
\begin{array}{rl}
Bob & \triangleq (vchat)friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\
Alice & \triangleq friend?y.y?q.y!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.y?answer \\
Carol & \triangleq oldfriend?y.\mathbf{y!12345} \\
System & \triangleq Bob \mid Alice \mid Carol \\
friend: & \mathsf{lin}\,?\,(\,\mathsf{lin}\,?\,\mathsf{string}.\,\mathsf{lin}\,!\,\mathsf{string}.\,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end} \\
oldfriend: & \mathsf{lin}\,?\,(\,\mathbf{lin}\,!\,\mathbf{string}).\mathsf{end}
\end{array}
\tag{2.2}
$$

In the Example 2.2 there is no communication safety since the channel $chat$ is supposed to be used to exchange three strings, and $Carol$ who was supposed to send the (last) answer to $Alice$ sends an integer instead. This code would be excluded from our type analysis, which could check that a conflict is present.

$$
\begin{array}{rl}
Bob & \triangleq (vchat)friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\
Alice & \triangleq friend?y.\mathbf{y!\text{“}Hello\ Bob,\ who\ is\ your\ friend\ ?\text{”}.y?q}.y?answer \\
Carol & \triangleq oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \\
System & \triangleq Bob \mid Alice \mid Carol \\
friend: & \mathsf{lin}\,?\,(\,\mathbf{lin}\,?\,\mathbf{string}.\,\mathbf{lin}\,!\,\mathbf{string}.\,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end} \\
oldfriend: & \mathsf{lin}\,?\,(\,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end}
\end{array}
\tag{2.3}
$$

In the Example 2.3 the communication channel $friend$ used by $Alice$ is not used like it is supposed to. $Alice$ tries to send a string before receiving a string, so the participants are not following a protocol, in particular the one specified by the type.

$$
\begin{array}{rl}
Bob & \triangleq (vchat)friend!chat.chat!\text{“}\texttt{Hello Alice!}\text{”}.chat?a.oldfriend!chat \\
Alice & \triangleq \mathbf{friend?y}.y?q.y!\text{“}\texttt{Hello Bob, who is your friend ?}\text{”}.y?answer \\
Carol & \triangleq oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.}\text{”} \\
Daniel & \triangleq \mathbf{friend?y} \\
System & \triangleq Bob \mid Alice \mid Carol \mid Daniel \\
friend: & \mathsf{lin}\,?\,(\,\mathsf{lin}\,?\,\mathsf{string}.\,\mathsf{lin}\,!\,\mathsf{string}.\,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end} \\
oldfriend: & \mathsf{lin}\,?\,(\,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end}
\end{array}
\tag{2.4}
$$

In the Example 2.4 the channel $friend$ is supposed to be used *linearly*, but it is not. The *linearity* of the session types assures *deterministic protocols*, avoiding races ($Daniel$ and $Alice$ on **friend?y**) and in addition to that, they do not use the channel $chat$ consistently which cannot happen neither in linear channels or in unrestricted channels. In summary it is impossible to specify a protocol like this because we don't know which "path" ($Daniel$ or $Alice$) will be "taken". Since protocols described by session types are deterministic one may talk about the several stages of the protocol as the "path" is known. Given this informal introduction to session types we now turn to the formal specification of the session type language.

2.2.1 Session types language

Figure 2.4 describes the syntax of the types. The first part of the language captures the communicating capabilities. We use the *p* to represent the *polarities*, the ! captures the output communication, the input capability is captured by ? and the τ captures a synchronisation pair. A system characterized by a τ message type is expected to have a synchronization in that message internally, while a system characterized by a ! or by ? message type is expected to interact with its external environment (either sending to it or receiving from it, respectively) to synchronize in that message. The types are divided into shared and linear, represented by the lin/un annotations. A shared type (unrestricted) supports multiple uses of the channel, where races are allowed, those allowing to support, for example, a "typology" of many clients and a server. The linear type (lin) captures linear usage of the channels, where no races are allowed and a sequence of linear deterministic communications must be followed.

$$
\begin{array}{rcll}
p & ::= & ! & \text{(Output)} \\
 & | & ? & \text{(Input)} \\
 & | & \tau & \text{(Synchronisation)} \\
 & & & \\
L & ::= & \mathsf{end} & \text{(No interaction)} \\
 & | & \mathsf{lin}\, p\, T\, .L & \text{(Session)} \\
 & & & \\
T & ::= & L & \text{(Linear)} \\
 & | & \mathsf{un}\, p\, T & \text{(Shared)}
\end{array}
$$

Figure 2.4: The syntax of types

A shared type $\mathsf{un}\, p\, T$ specifies a polarity p that captures a certain communicating capability, and a type T that describes the expected behaviour of the communicated channel. A linear type $\mathsf{lin}\, p\, T\, .L$ is similar, with the difference of the continuation L that determines the behaviour after the $\mathsf{lin}\, p\, T$ — this is possible in linear interactions due to existing a single path, being thus possible to specify the remaining interactions. A linear type terminates with end, meaning that no more interactions are to take place on that channel.

2.3 Session types using events

Session types with events add to session types the notion of event, this work [9] presented by Vieira and Vasconcelos was inspired in the notion of event from [4] and the approach presented in [7]. An event is an annotation added to the session types which allows to identify the communication event that is associated to the communicating actions described by the session types. The main idea is to introduce a notion of timestamps that

represent the abstract moment when each input or output action is supposed to happen, mainly allowing to causality-based relate the communication actions among them.

The events support the definition of a *strict partial order* that captures the dependencies between the communicating actions of the program. The goal is to assure the absence of deadlocks prior to the execution of the code, just by looking at the source of the program. With the complete notion of the overall order of the events, we are able to verify the progress property.

$$\begin{array}{l} \textit{Channel Types with events}: \\ x : e_1\, \tau\, \mathsf{Integer} \\ y : e_2\, \tau\, \mathsf{String} \end{array} \qquad (2.5)$$

$$\textit{Process}: \\ x!23.y!\text{“}\texttt{How old are you ?”} \mid y?question.x?answer$$

Example 2.5 shows a simple program that is deadlocked: we have a process with a parallel composition which uses two channels, channel y to ask a question and channel x to answer the question. This program is deadlocked because both threads are waiting for a synchronization that cannot take place: the process on the right hand side is waiting to synchronize on y while the process on the left hand side is waiting to synchronize on x. The fact that the other endpoints of channel x and y are not immediately active originates the deadlock — notice there is a cyclic dependency on the channel usages. Using the *event annotations* associated to the channels we may detect the cyclic dependencies that cause the deadlock, as we show in the following paragraphs.

For each synchronization we attribute distinct event annotations, for example the event annotations e_1 and e_2 correspond to the synchronization on channel x and y respectively. Inspecting the process on the left hand side of the process we conclude that the ordering of events is $e_1 \prec e_2$ since the interaction on channel x precedes the interaction on channel y — we use $e_1 \prec e_2$ to say that the event e_1 *happens before* event e_2.

On the right hand side of the process a different order of actions on the channels x and y is present and the ordering of the events is $e_2 \prec e_1$, because the channel y is used before channel x. This leads to a *cyclic dependency* ($e_1 \prec e_2$ and $e_2 \prec e_1$), allowing us to exclude the program as (potential) deadlock.

We turn to a familiar example, now focusing on the progress property.

$$\begin{array}{rcl} Bob & \triangleq & friend?y.y!\text{“}\texttt{Hello Alice!”}.y?a.oldfriend!y \\ Alice & \triangleq & (vchat)friend!chat.chat?q.chat!\text{“}\texttt{Hello Bob, who is your friend ?”}. \\ & & chat?answer \\ Carol & \triangleq & oldfriend?y.y!\text{“}\texttt{Hi Alice, my name is Carol.”} \\ System & \triangleq & Bob \mid Alice \mid Carol \end{array} \qquad (2.6)$$

Example 2.6 exhibits a communication similar to the Example 2.1 with the difference that this time it is $Alice$ that creates the private channel $chat$ and sends it to Bob. The process Bob receives the channel from $Alice$, uses the channel and then sends it to be used by $Carol$. In our approach, progress is also ensured in this kind of situation where processes interact on different channels (including received ones) in sequence.

Example 2.6 allows to introduce the notion of strict partial order, used to verify if the program has a well-formed communication structure (without cyclic dependencies). This order consists of a binary relation that allows to capture communication dependencies by establishing an order of the *events* associated to the communication actions. This order is strict (an event can't happen before itself $e_1 \prec e_1$), transitive and partial (since not all the events are related and are not be temporally related). The pairs in the relation represent a temporal dependency: the first event of the pair happens before the second event, i.e., $e_3 \prec e_1$ denotes that (e_3, e_1) is a pair of the relation which represents e_3 happens before e_1.

The different points of view of channel uses with the *strict partial orders of the abstract events*, from each participant are defined by :

$$friend : e_1 \,\mathsf{lin}\,!\,(e_2 \,\mathsf{lin}\,?\,\mathsf{string}.\, e_3 \,\mathsf{lin}\,!\,\mathsf{string}.\, e_5 \,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end} \vdash Alice$$

From the point of view of $Alice$ the order of events expected is $e_1 \prec e_2 \prec e_3 \prec e_5$, since the process first uses the channel $friend$ to send a channel (e_1) , then it receives a string from the channel $chat$ (e_2), sends a string (e_3) and receives another string (e_5) in this order.

$$oldfriend : e_4 \,\mathsf{lin}\,?\,(\, e_5 \,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end} \vdash Carol$$

From the point of view of $Carol$ the expected order of events is $e_4 \prec e_5$, since it first receives a channel on the channel $oldfriend$ (e_4) and then uses the channel to send a string (e_5).

$$\begin{aligned} friend &: e_1 \,\mathsf{lin}\,?\,(\, e_2 \,\mathsf{lin}\,?\,\mathsf{string}.\, e_3 \,\mathsf{lin}\,!\,\mathsf{string}.\, e_5 \,\mathsf{lin}\,?\,\mathsf{string}).\mathsf{end}\,, \\ oldfriend &: e_4 \,\mathsf{lin}\,!\,(\, e_5 \,\mathsf{lin}\,!\,\mathsf{string}).\mathsf{end} \vdash Bob \end{aligned}$$

From the point of view of Bob the order of events expected is $e_1 \prec e_2 \prec e_3 \prec e_4 \prec e_5$, since the process first receives from channel $friend$ the private channel and then from this order, sends a String (e_2), receives a string (e_3) and sends the private channel on the channel $oldfriend$ (e_4) which is then used to send a string (e_5).

The orderings of the events $e_1 \prec e_2 \prec e_3 \prec e_5$, $e_4 \prec e_5$ and $e_1 \prec e_2 \prec e_3 \prec e_4 \prec e_5$ are sound since gathering the relations we obtain the strict partial order $e_1 \prec e_2 \prec e_3 \prec e_4 \prec e_5$.

2.3.1 Session types using events language

Figure 2.5 describes the syntax language of the types using events. The language is very similar to the previous one, presented in the Figure 2.4. What makes this language different is the notion of event represented by an e. The language now allows to specify a relation between the communicating action and the timestamp, that represents the abstract time in which the action takes place.

We assume given an infinite set of events, ranged over by $e, e_1, \ldots$, used as event identifiers. A shared type $e\,\mathsf{un}\,p\,T$, specifies a polarity p that captures a certain communication capability, with an event e, that creates the association between the action and the notion of event. A linear type $e\,\mathsf{lin}\,p\,T\,.L$ follows the same lines with the difference that L

$$
\begin{array}{rcll}
p ::= & ! & \text{(Output)} \\
| & ? & \text{(Input)} \\
| & \tau & \text{(Synchronisation)} \\
\\
L ::= & \mathsf{end} & \text{(No interaction)} \\
| & e\,\mathsf{lin}\,p\,T\,.L & \text{(Session)} \\
\\
T ::= & L & \text{(Linear)} \\
| & e\,\mathsf{un}\,p\,T & \text{(Shared)}
\end{array}
$$

Figure 2.5: The syntax of types with events

represents the behaviour that takes place after $e\,\mathsf{lin}\,p\,T$. The type T describes the expected behaviour of the communicated channel.

2.4 Summary

This chapter gives an overview of the main concepts in background work. The π-calculus model introduced in Section 2.1 allows to focus on the communication between concurrent processes by modelling the interactions based on sending and receiving on channels. Session types presented in Section 2.2 describes programs protocol with a representation of the interactions among the participants in the communication. Each participant has its own role established in the protocol. With the expected interactions specified for each participant by the session types it is possible to identify systems that enjoy the *protocol fidelity* property. The session types with events introduced in Section 2.3 introduces the notion of an event, which allows to associate an abstract timestamp to each communication action. Building on session types with events we may develop a tool to statically verify the progress property.

Chapter 3

Static verification tool

In this chapter we describe how the static verification tool works and which input is needed in the verification process such as the variable declarations, order of the events and the process.

3.1 Input Specification

This sections explains the input that the programmer has to provide to use the static verification tool. The tool receives a file with extension ".p" which contents are illustrated in Listing 3.1 and described in the following paragraph.

The first element in this example is a type declaration defined in the first line by **Type** A = e1 **lin # boolean.end** which allows to define type abbreviations. The variable declarations specifies the channel names and their respective usage in terms of the session types with events; notice that the usage of channel a is defined by the abbreviation name A that represents a previously specified type declaration. According to type A, channel a has a synchronization of a message content with a boolean type, while the channel b will synchronize a message content with a string type. Each of the communication types has a correspondent event associated to their usage, in this case the synchronization in channel a is associated to timestamp e1 and the synchronization in channel b is associated to timestamp e2. The second element is the order which specifies the temporal dependencies of the events, which, in this example, specifies that the event e1 has to precede the event e2. The last element included in the input is the process. The process in this example is formed by a parallel composition which specifies both branches of the composition are simultaneously active. On the left side of the parallel composition we have an output from the channel a followed by an output in channel b, while on the right side accordingly we have an input in channel a followed by an input in channel b. This results in a synchronization on channel a and then another on channel b.

Listing 3.1: Input file example

```
Type A = e1 lin # boolean.end

TypingContext = (booleanVar : boolean;
                 stringVar : string;
                 a : A;
                 b : e2 lin # string.end)

Order = (e1<e2)

a!booleanVar.b!stringVar | a?result.b?result2
```

3.1.1 Typing Context declarations

The typing context declarations specify the name of each variable and their respective protocol. The declarations allow to create a type environment which stores the specified protocol type usage of each variable name. To declare the typing context in the program the user must write **TypingContext** = (name : protocol) where the first parameter defines the name to be associated to the variable and the second parameter specifies the session type protocol to be used by that variable. Multiple variables can be defined, just by using the ";" operator to separate the different declarations.

3.1.2 Order

In order to be able to verify the progress of a communication we need the partial order of the events associated to the session types and access to the order efficiently as part of the verification process.

The order specify a relation "happens-before" between the different events in the communicating program. The user must specify the order by writing **Order** = followed by the event relations using the operator "<" in the middle of two events i.e., **Order** = (e1<e2) means that event e1 happens before event e2. To split possible different sets of unrelated event relations we use ";", i.e. (e1<e2<e3<e4);(e5;e6) which says that the event e5 happens before event e6 but neither of them are related to the remaining events in the order as illustrated by Figure 3.1.

Further details from the order implementation and methods will be presented in the following chapter.

3.1.3 Processes

In this section we describe the different types of processes that can be used to specify a communication-centred program. The communication processes can be divided into channel creation processes, input processes, output processes, parallel compositions and in inactive process. Listing 3.2 will help us introduce the different types of processes.

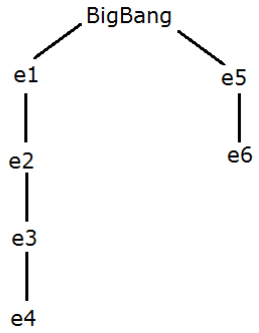

Figure 3.1: Order illustration example

Listing 3.2: Private session example

```
Type Chat = e2 lin # string.end

TypingContext = (server: e1 un ? (e2 lin ! string.end);
                 stringHello : string)

Order = (e1<e2)

*server?msg.msg!stringHello |
new chat : Chat in server!chat.chat?s |
new chat2 : Chat in server!chat2.chat2?s2
```

Channel creation

A channel creation allows to declare free variable names in the communication process to use as "private" names bound to the continuation process i.e. (**new** a : e1 **lin** # **boolean** **in** P) where a is the name of the channel and the channel usage is a linear synchronisation of a **boolean** type, associated to the event name e1, followed by the respective continuation process P which is the scope of the restricted name. A channel creation with a shared type can be used to create a private session with other process like Example 3.2 demonstrates. This example illustrates the interaction between two clients and a server. The communications are performed through a private session initiated from the clients side. The clients create a channel with the type usage (e2 **lin** # **string**.**end**) which means that a linear synchronisation of a message content with a **string** type takes place after the private session is established with the server. The created channel chat and chat2 respectively in each process and the following synchronisations allows each client and the server to communicate on a private session where they then interact. Notice that the server can handle an unlimited number of clients as the channel usage is specified as a shared input communication by the **un** ? annotation which is continuously waiting to receive a channel in *server?.

Input and Output Processes

The input and output processes can be divided in a linear communication or a shared communication depending on the channel usage specified. If the usage specified for a certain channel is a linear type, then the interaction must be carried out exactly once. If the specified type is a shared communication, then it supports various interactions like an open service, for example **un** ? (**lin** ? **boolean**) describes the usage of a shared channel that can be used zero or more times by a client, i.e., **un** ! (**lin** ? **boolean**). Notice instead that the message type specifies a linear synchronization **lin** (carrying a boolean value).

In Example 3.2 we have an open service made available by channel server, which is continuously waiting to receive a channel to perform the output of stringHello. From the client side point of view, each of the clients perform an shared output of a new private channel they just created on channel server to initiate a private session. After the private session takes place further interactions can happen, in this case each of the clients perform a (linear) input on the created channel (chat?s and chat2?s2), to receive the message content that the server sends on a (linear) output (msg!stringHello).

Parallel composition

The parallel composition $P|Q$ says process P and process Q are simultaneously (concurrently) active. In Example 3.2 the parallel compositions allows two client processes and one server process to interact.

Inaction process

This type of process represents the inaction in our type system, which to simplify is omitted from the process specifications. For instance we write $x!a$ instead of $x!a.\mathbf{0}$ to specify an output followed by inaction.

3.2 Verification process

A specification file is loaded in the tool to start the verification. Before starting the main verification of the communicating program, we perform a consistency check (see the description in Section 3.2.1) to all the variable declarations. The tool runs multiple rules over the specified processes, verifying if the sessions types protocol (with the event annotations) of each channel communication are concordant with their usage and order. After processing the file, the tool returns a boolean with the final result including an error message if the validation fails. If the result is positive, then the *progress* property is assured for that system.

3.2.1 Consistency check

The consistency check consists in confirming if all the associated events of the specified protocols in the variable declarations are consistent and in conformance with the specified

strict partial order. We have a problem of consistency when the events associated to a certain channel are not consistent to the order itself, for example if in the global order the event e_1 happens before the event e_2, it is not possible that there exists a channel with the events e_1 and e_2 associated where this relation ($e_1 \prec e_2$) is not respected. Another possible cause of failure is the presence of an illegitimate event which is an event that is non-existent in our order, causing a problem of conformity. If this verification fails the user is presented with an inconsistent event(s) type of error which is presented in Section 3.2.2.

3.2.2 Error Messages Types

It is necessary to provide some relevant information to the user when a communicating program fails in our static verification. In this subsection we will explain the different types of error which the user can be presented when the verification fails.

Syntax Errors

This kind of error happens when the user does not specify the input specifications correctly, hence the communicating program must be written accordingly to our grammar.

Unmatched communications

When some linear communication fails to be matched, this kind of error is presented with the respective unmatched channel type. For a communication to be matched in our type system, it must respect the predicate from Figure 4.1, used in our type system from [9] which do not accept type communications that specify pending communications actions (see the description for rule in Section 4.6.2).

Type was not fully used

This kind of error happens when a communication type protocol fails to be in a correct and expected state after a certain part of the verification. This kind of verification is also performed in the end of the main type verification to guarantee that no communication protocol in the type environment was left unused which would mean that an error had occurred. When this error happens, the user is informed with the channel and the respective type protocol that induced the error.

Inconsistent event(s)

When some event fails to be consistent with the actual global order of the events, this error is presented with the respective event failure. For an event to be inconsistent with the global order it must fail to be in a "correct position" according to the event marker of our strict partial order, which means that the event had already "happened in time" leading to a problem of consistency with the specified order. This kind of error can happen in any moment of the main verification to the communication processes, but can also happen before, since all the events associated to the variable declarations are verified to be in

concordance with the order in the consistency check like previously mentioned in the verification process.

Invalid type protocol

When a certain process contains an invalid communication type, this is presented to the user with the information of the incorrect channel type associated to the process with the respective erroneous type protocol.

Type of protocol not respected

This type of error is presented when the type protocol of a channel that is going to be output is not compatible with the specified type protocol in the type parameter of the output type. For example consider the following output specified by the type protocol (e1 **lin** ! (e2 **lin # string**.**end**).**end**) where it is expected to output a channel that will be used to make a synchronization of a message with a string in their content. If for some reason the channel type that is trying to be output have a type protocol that is not compatible with the type parameter of the output type (in this case e2 **lin # string**.**end**) then this error will be presented to the user with the respective informations regarding to the channel types.

3.3 Input Specifications Examples

In this section we present some example of communications programs and explain the response of the verification tool towards them.

The first example, represents a communicating program specified to perform two linear synchronizations using channel a, first a message with boolean type and after a message with a string type.

```
TypingContext = (booleanVar : boolean;
                 stringVar : string;
                 a : e1 lin # boolean.e2 lin # string.end)

Order = (e2<e1)

a?result.a!stringVar | a!booleanVar.a?result2
```

It happens that the events e1 and e2 associated to the type protocol, do not respect the global order (e1 $\prec$ e2) creating an inconsistent event(s) kind of error (Section 3.2.2), so the tool responds with the following error message :

```
The sessions types events are NOT consistent with the global order
(e2 < e1).
Channel a, with the session type (e1 lin # (boolean) e2 lin #
(string).end) does not respect the global order.
```

The following example represents a communicating program with two channels with one synchronisation each. The channel a has a type protocol which specifies a synchronisation on a message containing a string, while channel b specifies a synchronisation

message containing a boolean. The prefix type of channel a has the event e1 associated, while the event e2 is associated to the prefix type of channel b.

```
TypingContext = (booleanVar : boolean;
                 stringVar : string;
                 a : e1 lin # string.end;
                 b : e2 lin # boolean.end)

Order = (e1<e2)

b?result.a!stringVar | a?result2.b!booleanVar
```

The problem of this communicating program is that the usage of the channel b does not respect the global order (e1 $\prec$ e2), according to channel usage. Although on the right side of the parallel composition the channel usage is consistent with the order, on the left side, channel b does not respect the order when it tries to perform a synchronisation before the one from channel a, that have an event associated with a lower order. This problem creates an inconsistent event(s) kind of error but of a different kind regarding to the previous example, since this inconsistency results from an improper use of the channels and not by an inconsistency in the order of the events associated to a single protocol. The tool responds with an error stating that the event e1 associated to channel a is not of greater order (since it was supposed to be greater than event e2 that already had happened):

```
Process :
(a!stringVar)
Channel :a
Session type : e1 lin # (string) .end
Error : the event in the usage of a (e1) is of lesser order and
cannot be used in this context.
```

The following communicating program is based on an example from [9] with a slight variation which results in an error.

```
Type CHAT = e4 lin # string.e2 lin # string.end
Type sType = e6 lin # CHAT.end

TypingContext =
      (handle : e1 un ? (e2 lin ? string.end);
      stringHello : string;
      stringBye : string;
      service : e3 un ? (e4 lin ! string.e2 lin ? string.end);
      masterservice : e5 un ? (e6 lin ? (e4 lin ! string.
                      e2 lin ? string.end).end))

Order = (e3<e5<e6<e4<e1<e2)

new chat : CHAT in
  service!chat.chat?s.chat!stringBye |
  *service?y.new s : sType in masterservice!s.s!y |
  *masterservice?z.z?y.handle!y.y!stringHello |
  *handle?z.z?s
```

The communicating program in the example represents an open service being used by one client, where in the communication they are supposed to exchange some messages type contents containing strings. The service is represented by channel service, which delegates the communication with the client to channel masterservice that is supposed to send the stringHello to the client and delegate to channel handle to receive the response. Notice that all the three channels are shared communications prepared to receive multiple clients. The slight variation in this example regarding to the original is that the masterservice makes the delegation before sending the expected stringHello which results in a "type of protocol not respected" kind of error. The channel handle is supposed to receive a channel with a type protocol compatible with (e2 **lin** ? **string**.**end**). Instead of receiving a compatible type, channel handle receives channel y with the type protocol: (e4 **lin** ! (**string**). e2 **lin** ? (**string**).**end**). This incompatibility with the types results from channel masterservice trying to delegate channel y to channel handle before sending the stringHello channel (y!stringHello), specified by the unused prefix (e4 **lin** ! (**string**)) which results in the following error :

```
Process :
(handle!y.y!stringHello)
Channel :y
Session type : e4 lin ! (string)  e2 lin ? (string) .end
Error : y is not compatible with the expected parameter type of
channel handle (e2 lin ? (string).end).
```

The following example represents a communicating program with one synchronization of a message content containing a boolean type on channel a, followed by the channel creation of b with the protocol (e4 **lin** ! **boolean**.e5 **lin** ? **boolean**.**end**) that is associated to the type declaration bType, which is bound to the continuation process b! booleanVar.b?result.

```
Type bType = e4 lin ! boolean.e5 lin ? boolean.end

TypingContext = (booleanVar : boolean;
                 stringVar : string;
                 a : e1 lin # boolean.end)

Order = (e1<e4<e5)

a!booleanVar.new b : bType in b!booleanVar.b?result | a?result
```

This example allows to demonstrate an error caused by an unmatched type communication, that takes place when the channel creation from b tries to create the channel with an unmatched protocol type associated that do not respect our type system (according to predicate illustrated in Figure 4.1). Presented with this example our tool responds with the following error :

```
Process :
(new b in b!booleanVar.b?result)
Channel :b
Session type : e4 lin ! (boolean)  e5 lin ? (boolean) .end
Error : is not a matched linear communication.
```

The following example represents a program with a synchronisation on channel a. The variable declaration for channel a introduces the protocol e1 **lin** # (e4 **lin** ? **boolean**. **end**).**end**, where the parameter type e4 **lin** ? **boolean**.**end** means that the channel is used to receive a message containing a boolean value. The channel creation **new** c: cType **in** means that channel c is assigned the protocol defined by the type declaration cType.

```
Type cType = e4 lin # boolean.end

TypingContext = (booleanVar : boolean;
                 a : e1 lin # (e4 lin ? boolean.end).end)

Order = (e1<e4)

new c : cType in a!c.c!booleanVar | a?cChannel
```

The problem resulting from this example is that channel c is supposed to perform a synchronisation but fails to do so because there is no linear input on c to complete the communication. Notice that all it takes to resolve this problem is to add to the right side of the parallel composition a linear input on channel cChannel, as for example a?cChannel. cChannel?result. This problem results in the following error :

```
Process :
(a?cChannel)
Channel :cChannel
Session type : e4 lin ? (boolean) .end
Error : was not fully used.
```

Listing 3.3: Professor and Student Example

```
Type deliverType = e1 lin # string.e2 lin # boolean.end
Type privateType = e3 lin # string.e4 lin # boolean.end

TypingContext =
      (evalBoolean : boolean;
      workString : string;
      tpc : e5 un ? (e1 lin ? string.e2 lin ! boolean.end);
      evaluate : e6 un ? (e3 lin ? string.e4 lin ! boolean.end))

Order = (e5<e1<e6<e3<e4<e2)

new deliver : deliverType in
    tpc!deliver.deliver!workString.deliver?result |
    *tpc?x.x?studentWork.new private : privateType in
        evaluate!private.private!studentWork.private?eval.x!eval |
        *evaluate?z.z?work.z!evalBoolean
```

The example illustrated in Listing 3.3, represents a program with a process containing three concurrent branches in parallel representing a student, a professor and an assistant.

Figure 3.2 illustrates the channel communications between three processes. The communication starts when the process creates a private channel deliver to start a private session with the professor. The created deliver channel allows the student to send the work to the professor and then to receive the evaluation. After receiving the student work,

the professor delegates the marking to the assistant. To communicate with the assistant, the professor creates the channel private which allows the professor and the assistant to exchange the students work and the respective evaluation. After the professor receives the evaluation, he sends it back to the student using the previously created session (deliver). This program, which includes a sort of session interleaving which other approaches are incapable of addressing, is accepted by our tool.

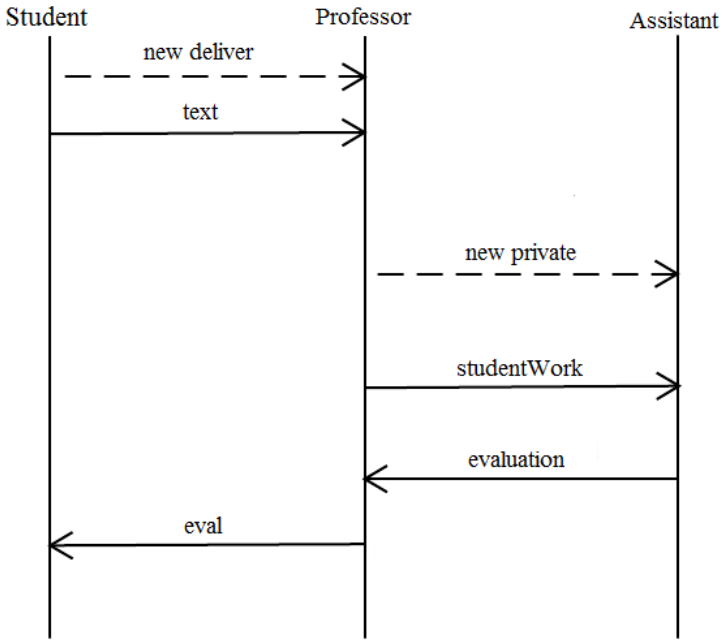

Figure 3.2: Message sequence chart for the work delivery example

The last example, illustrated in Listing 3.4, shows a program that deadlocks. The first reduction step invokes the service. The second reduction passes the reply channel as a parameter to the service, yielding the following process (types omitted) .

```
new reply in new request in
    request?y.reply!booleanVar.y?result |
    request!reply |
    *service?z.new reply in z!reply.z!reply
```

The request channel sent by service (in sub-process service!request) is used in the continuation process to receive the output capability (request?x), and then the input capability (request?y) from a channel with the protocol type : e4 **lin # boolean.end**. The problem happens because the input communication (reply?result) is performed in the continuation of the output communication (reply!booleanVar), so there can be no possible synchronization and therefore there is a deadlock.

Listing 3.4: Example of a deadlock

```
Type requestType = e2 lin # (e4 lin ! boolean.end).
                   e3 lin # (e4 lin ? boolean.end).end
Type replyType = e4 lin # boolean.end

TypingContext =
      (booleanVar : boolean;
      service : e1 un ? (e2 lin ! (e4 lin ! boolean.end).
                e3 lin ! (e4 lin ? boolean.end).end))

Order=(e1<e2<e3<e4)

new request : requestType in
  service!request.request?x.request?y.x!booleanVar.y?result |
  *service?z.new reply : replyType in z!reply.z!reply
```

Listing 3.5: Session types without event annotations

```
requestType = lin # (lin ! boolean.end).lin # (lin ? boolean.end).end
  replyType = lin # boolean.end
    service : un ? (lin ! (lin ! boolean.end).
              lin ! (lin ? boolean.end).end)
```

Notice that the session types without the event annotations given in Listing 3.5 would describe a classic well-typed session typed system, thus not excluding the deadlocked configuration. Instead, our tool excludes this system and presents the user with the following error:

```
Process :
(reply2?result)
Channel :reply2
Session type : e4 lin ? (boolean) .end
Error : the event in the usage of reply2 (e4) is of lesser order
and cannot be used in this context.
```

3.4 Users manual

In this section we explain how to use our static verification tool using the command line and using an Eclipse Plugin.

3.4.1 Running from the command line

To run our static verification tool using the command line it is necessary to follow this steps.

1. Download the `ProPi.jar` available in `http://download.gloss.di.fc.ul.pt/propi/jar/ProPi.jar`) which contains the verification tool ready to be executed.
2. Type an input specification (as demonstrated in Section 3.1) according to our grammar.
3. Run the jar file through the command line using the command :
   `java -jar ProPi.jar`.
4. Specify the specific path to the communicating program file (as illustrated in Figure 3.4) or a path to a folder containing multiple program files.

3.4.2 Using the Eclipse Plugin

To use our plugin in the eclipse the user must follow the following steps.

1. Download and install an Eclipse IDE.
2. In the Eclipse menu, go to **Help** menu and choose **Install New Software**.
3. In the **Work with** field introduce the update-site location :
   `http://download.gloss.di.fc.ul.pt/propi/update/`
4. Select the SDK Feature and press **Next**.
5. Restart Eclipse.
6. Create a **Project** or **Java Project**.
7. Create a file with extension ".p" and select **yes** when the editor asks to add the Xtext nature to your project.
8. Write a program on the created file and press **save** so the tool can check if there are errors like illustrated in Figure 3.3.

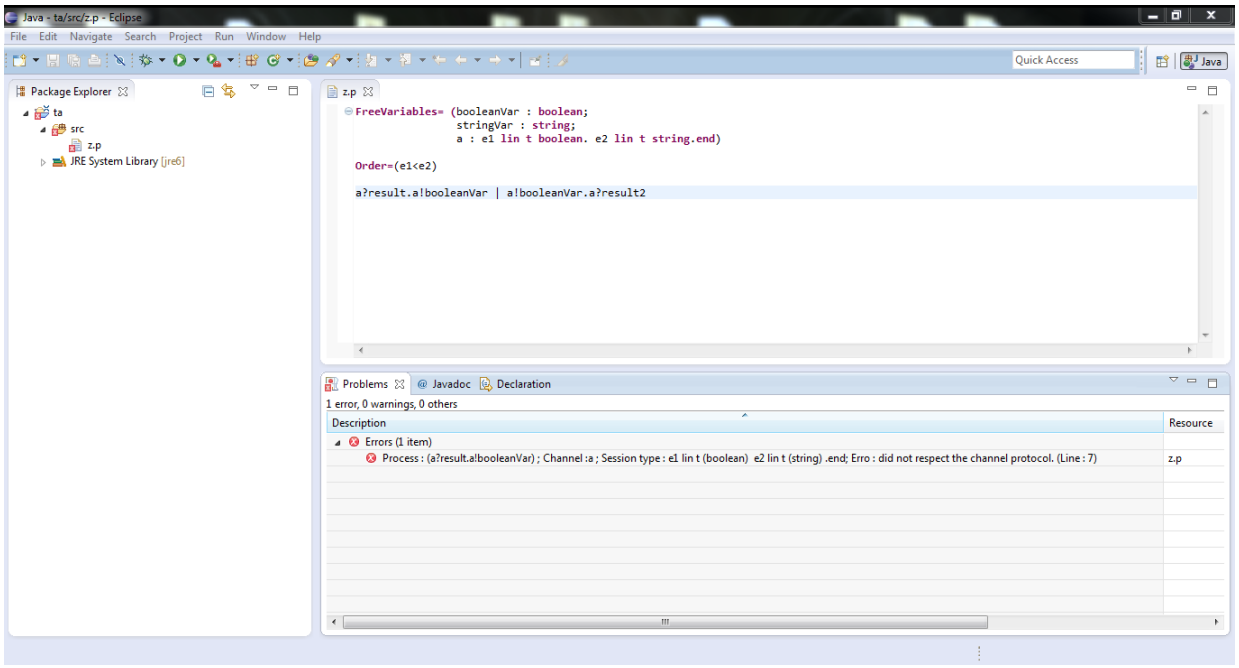

Figure 3.3: Eclipse with an example

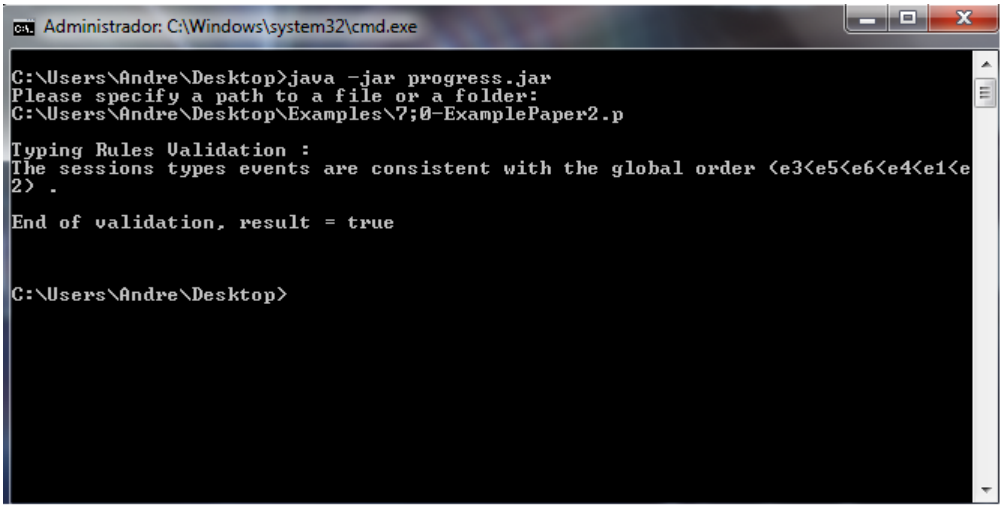

Figure 3.4: Command line menu

Chapter 4

Implementation

This chapter gives the necessary intuitions to understand how do we implemented the multiple elements necessary in our type checking procedure and also allow to show the main parts of the verification process, in which the verification rules identify well-formed communications processes permitting to distinguish communicating programs with progress. We highlight specific issues of the implementation that are distinguishing features of our work, such as how we implemented type extraction and how we validate linear channels are fully used.

4.1 Grammar

$$
\begin{array}{rcl}
DeclarationProcess & ::= & Declaration\ StringPartialOrder\ Processes \\
 & & \\
Processes & ::= & Process\ (|Process)* \\
Process & ::= & Input \mid ReplicatedInput \mid Output \mid \\
 & & Parallel \mid ProcessChannelCreation \\
Input & ::= & Variable\ ?\ Variable\ [.Process] \\
ReplicatedInput & ::= & *Variable\ ?\ Variable\ [.Process] \\
Output & ::= & Variable\ !\ Variable\ [.Process] \\
Parallel & ::= & (\ Processes\ ) \\
ProcessChannelCreation & ::= & new\ Variable\ :\ Type\ in\ Process \\
 & & \\
Declaration & ::= & TypingContextDeclaration \mid TypeDeclaration \\
TypingContextDeclaration & ::= & TypingContext = (Variable\ :\ Type \\
 & & (;Variable\ :\ Type)*) \\
TypeDeclaration & ::= & Type\ TypeVariable\ =\ Type \\
 & & \\
StrictPartialOrder & ::= & Order = (EventOrder(;EventOrder)*) \\
EventOrder & ::= & (Event(\prec\ Event)*?)
\end{array}
$$

4.2 Technologies

In order to develop our static verification tool, we used Xtext, which is an open-source framework for development of programming languages and domain-specific languages. This framework allow to write our grammar and to implement the verification mechanisms. After writing our grammar, Xtext allows to execute a code generator (Xtext artifacts) which derives the various language components using MWE2 (Modeling Workflow Engine 2). This code generation creates classes to represent multiple nodes of a generated abstract syntax tree. The created abstract classes allow to easily implement the verification rules which are then used to perform the static verification of progress in a specified communicating program written with our grammar. The framework also allowed to create an eclipse plugin based on the developed static verification tool permitting the user the ability to use our tool in an eclipse environment.

4.3 Verification process

In this section we explain how the verification process uses the input elements and returns to the user a validation response for a specified communicating program. The first step of the verification process is to obtain the variable type declarations, strict partial order and the process specified previously by the user. This first step is demonstrated in a Java-like pseudo-code Listing 4.1, starting by passing the DeclarationProcess to a visitor which within the visitor proceeds by passing the variable declarations and the order to other respective visitors.

Listing 4.1: Verification Process Pseudo-Code

```
check(P p) {
  DeclarationProcessVisitor visitor =
  new DeclarationProcessVisitor(javaValidator,symbolTable,order);
  visitor.doSwitch(p);

  DeclarationProcess dp = visitor.getDeclarationProcess();
  StrictPartialOrder spo = dp.getOrder();
  Processes procs = dp.getProc();

  TypingRulesValidator trv =
  new TypingRulesValidator(javaValidator,symbolTable,order);

  boolean result = trv.mainTypeCheck(symbolTable,order,process);
```

The process will then be handled by mainTypeCheck which is presented in the pseudo-code from Listing 4.2. This method receives all the specified elements and has the work to return a positive or negative boolean result depending of the verification of progress. Through the various stages of the verification of progress, the protocols usage for each channel changes via updates to the type environment, that thus records the actual state of all the channel protocols. Notice that channel creations and bound variables of input communications also result in updates to the type environment. Before the main process verification, a consistency check is performed (as described in Section 3.2.1) to assure that all the variable declarations are consistent and in conformance with the global order.

If the consistency check fails, the verification stops and a negative boolean result is returned. In the opposite case the verification proceeds and the process is carried over to our ValidationSwitch which will handle the different types of processes within the main process and send each one to the respective verification rule. The Java code in Example 4.3 demonstrates how the switch proceeds with the processes and how it handles the linear and unrestricted input processes separately. The switch returns a positive result if all the processes pass the verification rule with no exception. At the end a last verification is performed by checking if all the variables used in the communicating program end in a correct state, in other words, it is assured that all the typing context variables are fully used, which means that for all the linear type communications an **end** type protocol must be stored and for unrestricted type communications just shared output types.

Listing 4.2: mainTypeCheck Pseudo-Code

```
vSwitch = new ValidationSwitch(javaValidator,symbolTable,order);
boolean mainTypeCheck(symbolTable,order,Processes,freeVarsKeys) {
    boolean result = false;
    if(conformsCheck(symbolTable,order)){
      result =  vSwitch.caseProcesses(p);
      if(result)
        result = freeVarsconformsCheck(table,order,freeVarsKeys);
    }else{
      System.out("Not consistent");
      return false;
    }
    return result;
  }
```

Listing 4.3: Validation Switch Processes and Input Java Code

```
  @Override
  public boolean caseProcesses(Processes proc) {
    boolean result = true;
    Iterator<Process> it = proc.getProcs().iterator();
      while (it.hasNext()) {
        result &&= caseProcess(it.next());
      }
    return result;
  }

  @Override
  public boolean caseProcess(Process p) {
    if (p == null){
      return this.tpr.tInact(table, order, p);
    }
    return super.doSwitch(p);
  }

  @Override
  public boolean caseInput(Input p){
    boolean result = false;
    result =  this.tpr.tLIn(table, order, p);
    return result;
  }
```

```
@Override
public boolean caseReplicatedInput(ReplicatedInput p){
  boolean result = this.tpr.tUIn(table, order, p);
  return result;
}
```

A fundamental notion of our rule implementation is that the verification steps extract from the type environment only what they need, leaving the remaining in the environment to be used by the remaining processes. So it is necessary in the end to assure that all the typing context variables were fully used by verifying that each one ends in a correct state and without unused resources. This approach is an added value to our implementation since it is a different and more efficient way to address the implementation of the rules with respect to other known implementations of session types. We had to adapt the "mathematical view" of the rules to a context of implementation where managing the type environment is more operational and also attending to efficiency concerns. A simple example of this "mathematical abstraction" that we had to deal with was the concept of "exists an element" which in the mathematical environment does not have to be "managed" in the same way, since we had to really find the exact element in the implementation context. Our solution to this problem was to use an unique type environment which changes the internal state according to the variable usages in the verification steps, assuring that in each step the resources "extracted" are just the necessary ones for that verification rule, leaving the rest of the type environment unchanged for the next verification steps. This also avoids the necessity of creating multiple copies of the type environment with different states to each rule, which is more inefficient.

4.4 Type Environment

To implement the type environment we chose to use a symbol table which has an hash table data structure, using the channel name to create the symbol to be used as the key for the type protocol value. This corresponds to a standard choice to represent key value pairs, where we also account for key replacement to support name clashes (when two bound names have the same syntactic identity).

4.4.1 Methods

In this subsection we present the main necessary methods of the symbol table.

- **void** put(Symbol key, TypeInterface type) - this method is responsible to add the symbol key to the symbol table, with the respective TypeInterface type associated.
- **void** remove(Symbol key) - this method is responsible to remove the key symbol and the respective TypeInterface from the symbol table.
- TypeInterface get(Symbol key) - this method returns the TypeInterface associated to the key symbol on the symbol table.

- **boolean** containsSymbol(Symbol key) - returns if the symbol table contains the symbol key.
- **public** Set<Symbol> getKeys() - returns a set of all the keys in the symbol table.
- List<Pair<TypeInterface,TypeInterface>> linearTypeSplit(TypeInterface type) - this method returns a list with pairs of TypeInterface from a performed linear type split of parameter type.
- List<Pair<Symbol, TypeInterface>> removenonun() - remove all the linear type communications from the symbol table and set unrestricted communications with an output polarity, returns a list of pairs with the symbols and respective TypeInterface of the original elements to be then restored.
- **void** restoreTable (List<Pair<Symbol, TypeInterface>> list) - restores to the symbol table the elements from the parameter list returned from the previous rule removenonun().

4.4.2 Example of type environment updates

The type extraction is performed by the output rules, which have to assure that the parameter type of the output channel is compatible and can be extracted from the type protocol of the channel that is object of the communication. To further clarify how each verification process step extracts from the type environment only what is needed, we reintroduce an example similar to Example 3.2 (with just one client), which will allow us to show the changes in the the type environment state through the verification steps.

```
Type Chat = e2 lin # string.end

TypingContext = (server: e1 un ? (e2 lin ! string.end);
                 stringHello : string)

Order = (e1<e2)

*server?msg.msg!stringHello |
new chat : Chat in server!chat.chat?s
```

$$\begin{aligned} stringHello &: string \\ server &: e1\,un\,?\,(e2\,lin\,!\,string.end) \end{aligned} \tag{4.1}$$

Before the verification starts the type environment has the following state represented by the illustration 4.1. The represented type environment have 2 variables with the respective protocol types. The $stringHello$ has a *string* primitive type protocol, while the channel $server$ has a prefix shared input type. The first two main verification steps applied to the communicating program are performed by the rules (T-UIn) and (T-LOut) (further explained in Section 4.6) to the shared input process *server?msg and to the linear output process msg!stringHello respectively.

After this first two verification steps the type environment has the state represented in the illustration 4.2. Channel $server$ swaps the "?" annotation symbol for the wildcard symbol annotation "_" (further explained in Section 4.4.2), and channel msg is added with the type protocol of the type parameter from $server$, which is consumed by msg! stringHello, leaving an **end** type in the type environment (further omitted). Notice that rule (T-LOut) must ensure that it is possible to extract the parameter type of the protocol type of channel msg, which was a *string* from the type protocol of $stringHello$, also a *string* type.

$$\begin{aligned} stringHello &: string \\ server &: e1\, un\_(e2\, lin\, !\, string.end) \\ msg &: end \end{aligned} \tag{4.2}$$

The third main verification step is performed to the channel creation **new** chat : Chat **in** server!chat.chat?s by the rule (T-New), which results in the type environment state represented by the illustration 4.3 in which the new channel $chat$ is added to the type environment with the type protocol (e2 **lin # string**.**end**).

$$\begin{aligned} stringHello &: string \\ server &: e1\, un\_(e2\, lin\, !\, string.end) \\ chat &: e2\, lin\, \#\, string.end \end{aligned} \tag{4.3}$$

The next verification step is performed by rule (T-UOut) to the communication process server!chat. In this verification step the shared output rule has to assure that it can extract the parameter type (e2 **lin ! string**.**end**) of channel $server$ from the protocol type of channel $chat$ (e2 **lin # string**.**end**). The extraction is possible because the protocol type (e2 **lin # string**.**end**), which represents a synchronization of message content with a *string* type can be separated in the individual input and output communication : (e2 **lin ? string**.**end**) and (e2 **lin ! string**.**end**), so the extraction is possible and the remaining unused protocol type (e2 **lin ? string**.**end**) is left in the type environment to be further used by the next process, as the illustration 4.4 demonstrate.

$$\begin{aligned} stringHello &: string \\ server &: e1\, un\_(e2\, lin\, !\, string.end) \\ chat &: e2\, lin\, ?\, string.end \end{aligned} \tag{4.4}$$

The following verification step completes the main verification process of the communication processes. The process chat?s is verified by rule (T-LIn) which uses the remaining protocol e2 **lin ? string**.**end** that was left in the type environment to validate the process, leaving an **end** type protocol in the channel $chat$ meaning that no further interaction will take place on that channel. The final state of type environment is illustrated in 4.5.

$$\begin{aligned} stringHello &: string \\ server &: e1\, un\_(e2\, lin\, !\, string.end) \\ chat &: end \end{aligned} \tag{4.5}$$

Table 4.1: Extraction Table

Full type	Extracted type	Remaining type	Notes
_	!	_	Polarity stays the same
_	?	_	Polarity stays the same
?	!	?	Polarity stays the same
?	?	_	Polarity changed to _
!	!	!	Polarity stays the same
!	?	Return False	Not possible to extract ? from !
!	_	Return False	Not possible to extract _ from !
?	_	Return False	Not possible to extract _ from ?
_	_	Return False	Not possible to extract _ from _

WildCard Polarity

The wildcard polarity is used in our type system in the context of shared communications. A shared type communication, like a linear type communication, can be divided in input and output communications. But while a linear type must be used strictly once, an input shared type describes a reception of a message one or more times and an output shared describes an emission of a message zero or more times. The wildcard polarity allow us to represent the "zero or more times" that the channel can still be used after the first one which is not possible with the regular "?" annotation. So when a shared input communication is used for the first time, the remaining type polarity is "_", to be used by the following verifications, which ensures that the channel was already used at least once. This auxiliary notation is a necessary addition to the theoretical system in which our work is based, allowing us to type check channel extractions from the type environment when synchronising messages containing shared communications like further explained using a table.

The extraction table illustrated in Table 4.1, allows to explain how the polarities can be extract in the shared communications context. In the first two lines, the "_" polarity means that the service was already used and that it is available to be further used zero or more times, so we can extracted both the "!" (to use the service) and "?" (to offer the service), the remaining polarity is the same since it continues to be available to be used zero or more times. In the third and fourth line of the table, the polarity "?" means that the service is still available to be used at least once and possibly more times, also allowing to extract both the "!" and "?" polarities like the previous case. The remaining polarity in the third line remains the same because the "!" just uses the service, while on the fourth line the service is offered so it changes to the wildcard polarity meaning that the service was already offered at least once. On the fifth line the "!" polarity means that it allows to output unrestrictedly on that service zero or more times, so we can just extract a "!" polarity, but can not extract a "?" polarity like the sixth line illustrates since the "!" polarity is just the use of the service an not the offer of the service. All the other lines represents extractions that are not possible. Both the "?" and "!" polarity can not assure that the service is still available zero or more times, and the last case can never happen since it is only possible to extract an input or an output, but not a wildcard. So all this last cases receive a "return false" mark in the table illustration.

4.5 Order

The access to the order must be efficient since it is necessary to check all the events and their relation to assure that the events are consistent with the protocols usage. The challenge was to assure the verifications with the lowest number of accesses to the order, avoiding costly changes in the data structure every time we need to change the order itself through the verification. To implement the order we created a "static order" class representation where the overall order does not change but the node reference explained below does. Notice that the overall order can have a potentially big dimension, which means that creating a new order every time that the verification progresses and the obvious need to "discard in time" already used events was not a viable option. With our approach we can get a better efficiency managing the order to the purpose of this work.

To accomplish this we use a node reference which is the only attribute that changes during the verification. This node reference is used to represent the root of the current order (regardless if such an element is not the root of the entire order) at any moment of the verification and is easily changed as necessary, allowing us to manipulate sub-orders as needed. This sub-orders allow to efficiently validate different parts of the program through the verification process. The order class has a hash table which allows to access the event nodes by a key which is the name of the event. Each event node has the list of his parent nodes, hence, each event knows the event nodes which precede them, that therefore represent the order that must be respected by the communicating program. To represent the first event that happens before any other event we have created a "virtual" event and named it *BigBang*. This order representation creates a "forest" with one or more tree of event nodes, since not all the events have to be related, and all this possible tree of events have their first event preceded by the unique event *BigBang*, thus creating in fact a single tree of events. With this approach we have the possibility to check if a certain type specification with event annotations is consistent with the global order without any significant change to the order, besides the node reference which can be easily restored. Since the reference that marks the current event can change during the verification, there is no need to in fact "update" the overall order through the verifications. To verify if a certain event belongs to the order at a certain state of the verification, we use the method containsEvent which returns a boolean value. This kind of verification is performed by recursively checking the list of parents, until finding the reference that marks the current node. In the case of the node reference is not reached through the recursion, this means that the event does not belong to the current order as required and hence the verification fails.

Before starting the static verification through the processes in which the events are consequently verified recursively, a basic verification is performed first, to assure that all the defined session types have legitimate events associated and in consonance with the global order.

4.5.1 Methods

In this subsection we present the main necessary methods used to implement the order.

- **void** addBunch(List<EventRelation> list) - this method acts like a constructor, adds the event BigBang and the remaining event relations in the list , i.e. the relation (e1,e2) where event e1 is added to the parents list of event e2.
- **void** restoreMarker(EventNode marker) - restores the marker to the received EventNode parameter marker.
- EventNode getRoot() - returns the current EventNode marker reference in the order.
- EventNode changeRoot(Event event) - changes the current EventNode marker reference to the EventNode associated to the specified event and returns the old previous marker reference.
- **boolean** containsEvent(Event e,**int** parentDistance) - returns if the order contains the event e. It is a recursive method, in which the parentDistance parameter is used to assure that event e is at least "one event distance" from event marker reference.
- **boolean** conforms(TypeInterface t) - returns if all the events in TypeInterface t are contained in the order using the previous method containsEvent.
- **boolean** conforms(SymbolTable table) - returns if all the TypeInterface in table are contained in the order using the previous method for TypeInterface in table.

4.6 Rules

In this section we explain how we implemented the verification rules and what is their role in the verification process. All the rules receive a symbol table with the type environment, the order and the process itself. The rules (T-LIn), (T-LOut), (T-UIn) and (T-UOut) have a similar skeleton with individual differences that will be described later. Listing 4.4 represents the pseudo-code of this skeleton with the generalized common steps of the verification rules.

The first steps illustrated in the skeleton consist in acquiring the process symbols and the respective protocol types using the symbols as the keys to access to the protocol values stored in the symbol table. In the case the rule is an input or a replicated input, the previous type associated to that symbol (if any) is stored. The next steps consist in verifying if the protocol type associated to the process has a legitimate type for the respective rule, for example the rule (T-LOut) can not receive an input type associated to the channel of the process, (if this happens an error message is presented). After this first verification, the associated event is also confirmed to be in concordance using the method containsEvent, if this verification fails, the user receives an error message with the information of the event causing the problem. The following verification is represented by MainVerification where the differences between the different rules lie. After the main verification, in the case of an already existing protocol type associated to that symbol, the type is restored in

the type environment (so as to support bound name clashes). At the end the old marker is restored to restore the order to a previous state; notice that the order changes within the main verification in each rule and is restored to the stored previous marker at the end (outside the main verification).

Listing 4.4: Common Skeleton Pseudo-Code

```
boolean SkeletonPseudoCode(symbolTable, order, process) {

  boolean result = true;

  symbols = getProcessSymbols(process);
  types = getProcessTypes(symbols,symbolTable);

  if(Input || ReplicatedInput)
    oldType = savePreviousType(symbol,symbolTable);

  if(checkIsCorrectType(types)){
    if(!order.containsEvent(type.GetEvent())){
      setError("Inconsistent event error");
      result = false;
    }
  }else{
    setError("Invalid type protocol error");
    result = false;
  }
  if(result){
    MainVerification();

    if((Input || ReplicatedInput)){
      if(existsOldType())
        table.put(symbol, oldType);
      else
        table.remove(symbol);
    }
    order.restoreMarker(oldMarker);
  }
return result;
}
```

4.6.1 T-Inact Rule

This rule represents the type check of the inaction, which is represented by an **end** in our type system. The goal of this rule is to type check all the inaction processes. The channel usage of a inaction process is none, in concordance with our type system the inaction process do not use anything in the type environment, so the verification always succeeds and leaves the type environment untouched.

4.6.2 T-New Rule

This rule has the role to type check the channel creation processes. Which verification steps are illustrated in Listing 4.5. The first verification performed is to check if the

created channel has a type matched communication, since we do not allow unmatched communications of restricted channels in our type system. We say that a type is matched if it does not specify pending communications actions, so we exclude shared outputs and linear communications that are not synchronisations as described in the predicate by Figure 4.1 from [9]. A matched linear communication is captured by the annotation # in the protocol which means that there is a synchronisation to take place between two processes. If the type of the new channel is matched, then it is added to the symbol table. Then using a switch the verification proceeds to the continuation process to the respective rule, which then returns the recursive result of the type checking of the continuation process. If the result is positive, then we check if the type of the new channel was completely used to validate the process. At the end we restore the channel type of the used name to the previous one (if any). This gives the possibility to overlap channel names in different contexts without losing the content of the previous channel usage.

Listing 4.5: New Channel Pseudo-Code

```
Boolean tNew(symbolTable, order, ProcessChannelCreation process){

  Type channelType = visitor.caseType(process.getType());

  if(!type.matched(channelType)){
    setError("Unmatched communication error");
    result = false;
  }

  if(result){
    table.put(channelSymbol, channelType);
    result = Switch.caseProcess(process.getProc());
    if(result){
      if(!checkTypeFullUse(channelType, symbol, table)){
        setError("Type not fully used error");
        result = false;
      }
  }

  if(channelExisted)
    table.put(channelSymbol, oldType);
  else
    table.remove(channelSymbol);

  return result;
}
```

$$matched(T) \triangleq \begin{cases} matched(L) & \text{if } T = e\,lin\,\tau\,T_1.L \\ true & \text{if } T = end \text{ or } T = e\,un\,?\,T_1 \\ false & otherwise \end{cases}$$

Figure 4.1: Matched predicate

4.6.3 T-Par Rule

This rule proceeds each one of the individual process in the parallel composition to the verification. Notice that the verification of the first process can use resources from the type environment that will be further used to verify the second process.

4.6.4 T-LIn Rule

This rule is applied to the linear input processes. Following the skeleton Figure 4.4, the first verification is performed to the channel type of the process: in this case only a linear synchronisation or a linear input are accepted. After validating the channel type we verify if the event associated to the process is in conformance with the actual state of the global order of the events. If the event is not valid, then a error message is presented and the validation fails; the same applies if the channel type is not the expected. After this first tests the main verification of this rule (illustrated in Listing 4.6) starts: first we add the channel y and the respective type to the type environment. The current event is marked as root, which is essential to validate the soundness of following events, assuring that the continuation process just has events of greater order. When the root is changed, the method also returns the previous root marker which will allow to restore the order to the previous state, so the following verification can proceed. The following verifications depends on the channel type where within the method tryCombinations different combination of type splinting are tested. If the protocol usage of the channel is a linear synchronisation then it is necessary to open the type to the correct input and output types separately; to find the correct split it is necessary to try to type check this different combinations recursively, always having in consideration the continuation of the linear type. On the other hand if the channel type is a linear input communication then the split is more simple. In this case the split consists into the already known input type and an ***end*** type. Notice that a split separates a matched channel type in output and input capabilities only, so splitting happens only once for each matched channel type. In both cases a recursive validation is performed inside the method to assure the correctness of the splits in the communicating program. The continuation type of the left side of the split is assigned to the channel type to assure that the performed type split type checks, if so, the channel type is assign with the right side of the split and the verification continues. After this recursive verifications we confirm the full use of the channel protocol, assuring that the protocol was complete. In the end we restore the channel type to a previous one if the same channel name already existed before as we explained in the previous rule. The order marker is also restored to the oldMarker, which was saved before starting the recursive validations, so the order can be maintained valid after the necessary changes in the event marker through the recursion.

Listing 4.6: Linear Input Main Verification Pseudo-Code

```
MainVerification(){
    table.put(ySymbol, xParamType);
    oldMarker = order.changeRoot(inputEvent);
    if(!tryCombinations(xSymbol,ySymbol,table,p,error){
      setError("Type of protocol not respected error.");
      result = false;
    }else
      if(!checkTypeFullUse(xParamType,ySymbol,table)){
        setError("Type not fully used error.");
        result = false;
      }
}
```

4.6.5 T-LOut Rule

This rule is applied to all linear output processes. The initial part of the verification follows the skeleton in Listing 4.4. The first verification performed to the channel type is analogous to all other rules. In this case the process type is assured to be a linear synchronisation or a linear output. If the process type is valid, then the event associated to the process is verified to be in concordance with the global order using the method containsEvent, if the validation fails because of the process type or an invalid event situation, then respective errors are presented to the user. Also similarly to the previous linear rule, the actual event is marked as root to save the current state of the order and the core of the verification starts, like illustrated by Listing 4.7.

First we need to assure that the parameter type of the channel to be output (y) conforms to the specified channel protocol. To perform this verification we use canExtract method to compare the type protocol of the channel with the type parameter of the protocol and verify if the first can be extracted from the second. The type environment is updated consistently if the extraction succeeds or an error message presented to the user if it fails. If the previous verification succeed, a recursive validation starts through the method tryCombinations. The recursive validation through the processes is necessary to verify if "some path" validates the communication process. To do this do we need to open the type in two parts, to the correct input and output types separately in the case we are dealing with a linear synchronization, notice that in most cases multiple combinations are necessary to cover all the possibilities. In the case we are dealing with a linear output communication, they split into a single output and an ***end*** type, always having in consideration the continuation processes of the the type in both of the cases which are also validated too through the recursion. If no valid possible combination of type splits exist to validate the communication process, then the user receives an error message explaining that the protocol specified for that certain communication channel is not respected.

After this validation, one more verification is necessary. We need to verify if all the events associated to the channel protocol that is output, are contained in the actual order state. To verify if all the events from the channel are legitimate to our order, we get the type protocol from the channel and test all the events associated with containsAll method, which calls the containsEvent individually for each event. Notice that if the channel is a

shared type communications then just exists one event to be verified. In the end of this rule the order marker is restored to the oldMarker, which was maintained before starting the recursive validations like in the previous rule.

Listing 4.7: Linear Output Main Verification Pseudo-Code

```
MainVerification(){
    oldMarker = order.changeRoot(outputEvent);
    if(!canExtract(table.get(ySymbol),xParamType,table){
      setError("Type of protocol not respected error.");
      result = false;
    }
    if(result){
      if(!tryCombinations(xSymbol,ySymbol,table,p,error)){
        setError("Type of protocol not respected error.");
        result=false;
      }else
        if(!containsAll(table.get(ySymbol),order)){
          setError("Inconsistent event(s) error");
          result = false;
        }
    }
}
```

4.6.6 T-UIn Rule

This rule is applied to all the shared input processes. The first part of the verification is analogous to the other rules, where the channel type is verified to be in agreement with the type of process and the event associated is confirmed to be in conformance with the global order. In this case for the channel type to be in agreement it must be a shared type communication with an input or wildcard polarity with a corresponding event associated. If some of this verifications fail, the user receives an error message with the respective error information.

The following steps of the verification process to be explained are illustrated in Listing 4.8. Within the main verification, some temporarily changes are performed to the type environment. First we need to remove the channel type from the type environment by removing the protocol associated to the x symbol from the symbol table. Then all the linear type protocols in the type environment need to be momentously removed (set to an ***end*** type), and all the remaining shared communications set to the output polarity ("!") to proceed the validation. Notice that all of the symbol and types from the linear and shared communications in the symbol table are stored before the changes, with the purpose of being restored after this validation. This changes are necessary because only unrestricted communications can be present in the type environment to perform this verification, since it is not allowed to use any linear resources in the continuation process besides possibly on the received channel. Also the reason why the the shared communications polarities are necessary to be changed while the verification is being performed, is because we can not allow other services offers to take place (other input shared) in the continuation process of that shared communication.

After this changes in the type environment, the parameter type of the channel type is added to the type environment with the y symbol and the order marker is changed to the input event associated to the channel type, always storing the old marker to be restored in the end of the verification. Subsequently the next process (associated to the actual process) is proceeded to the respective rule, returning the result of the recursive verification of the following processes. If the result is positive then we proceed to verify if the protocol that was associated to the y symbol was fully used which means to verify if it terminates in a correct and expected state. The last part of this verification consists in restoring the changes previously performed. All the linear and shared type communication previously stored return to the type environment, the channel type polarity is changed to a wildcard polarity (further explain in Section 4.4.2) and restored to the type environment. At the end the order marker is also restored to the previous stored old marker. Notice that analogously to the (T-New) and (T-LIn) Rule, in the case of previously existing a channel using the same channel name, the first one is restored to the respective previous type which was saved in the beginning of the rule.

Listing 4.8: Shared Input Main Verification Pseudo-Code

```
MainVerification(){
    table.remove(xSymbol);
    savedTypesList = table.remove_non_shared();
    table.put(ySymbol,xParamtype);
    oldMarker = order.changeRoot(inputEvent);

    result = Switch.caseProcess(p.nextProc());
    if(result){
      if(!checkTypeFullUse(xParamType,ySymbol,table)){
        setError("Type not fully used error.");
        result = false;
      }
      table.restoreTable(savedTypesList);
      table.put(xSymbol,xType.setPolarity("_"));
    }
}
```

4.6.7 T-UOut Rule

This rule is applied to all the shared output processes. To pass through the first verification the channel type of the process must be a shared output communication and have an event associated in concordance with the global order. The following steps of the verification process are illustrated in Listing 4.9. After passing the first verifications, the x symbol is removed from the symbol table. The channel usage is rule out in the continuation to exclude processes that offer an input in the continuation of an output. Similarly to the linear output rule, using canExtract method, it is confirmed that it is possible to extract the channel y type protocol in the type environment from the parameter type specified in channel x protocol, so the type environment can be updated. If the operation is not possible then an error message is presented to the user explaining that the channel do not respect the expected protocol (further details of the type environment updates in Section 4.4.2). After this point we change the event marker to the event associated to the channel type,

always storing the previous marker to restore it later. The next step is to verify if all the events associated to the channel y protocol are legitimate and in respect to the present global order of events, this step is accomplished using the containsAll method that uses the containsEvent method for each event, once again an error message is displayed to the user in regard to the order if some event is not in concordance. After this step the next process is proceeded to the the respective rule and a boolean result is received recursively from the following verifications. We now can add the respective x channel protocol back to the type environment and restore the event marker to the previous event marker.

Listing 4.9: Shared Output Main Verification Pseudo-Code

```
MainVerification(){
    table.remove(xSymbol);
    if(!canExtract(table.get(ySymbol),xParamType,table)){
      setError("Type of protocol not respected error.");
      result = false;
    }
    oldMarker = order.changeRoot(outputEvent);
    if(!containsAll(table.get(ySymbol),order)){
      setError("Inconsistent event(s) error.");
      result = false;
    }
    result = Switch.caseProcess(p.nextProc());
    table.put(xSymbol,xType);
}
```

Chapter 5

Conclusion

This thesis presents a type checking procedure based on session types complemented with the notion of event. Session types allow to describe the interactions of each channel, while the events allow to guarantee that the flow of the program follows a sound ordering. To develop this work it was essential to study and learn a new concepts that took time to become clear, more precisely an introduction to the π-calculus [5, 6], the use of session types in a linear π-calculus [8] and the use of session types with events [9].

Our work consisted in developing a tool that allows to statically guarantee fidelity and progress for message passing concurrent systems with point-to-point communications. We created a grammar which permits to specify the session types (with the events annotations) and the process interactions. The tool verifies the specified program through a set of rules, allowing to distinguish concurrent systems that respect the specified protocol and achieve progress. Our grammar is separated in three main elements, the variable declarations which specifies the variables and respective protocols types which are the base to create the type environment, the order of the events which uses the "happens-before" relation to define an order between events and the process containing the interactions that form the communications.

The manipulation of type environment was a challenge, which resulted in a different kind of approach. The main challenge was the necessity to follow the mathematical rules and deal with the mathematical abstraction inherent to them. Our approach distinguishes itself in the way that the updates are performed, since each process uses just what it needs in each verification step, leaving the rest to be used by the remaining processes. This allows to use always the same environment passed on from rule to rule and avoid having more than one environment with different states. Although there are other works that follow the same approach, we believe our implementation, to some extent, is more streamlined at the level of type extraction.

Another challenge was the implementation of the order relation, which had the efficiency as main concern, since the verification procedure performs multiple operation on the order relation. Our solution was to create a "static order" which has a reference marker that is the only element that changes, and which allows to create different sub-orders. This implementation allows to avoid the need to restructure the order throughout the verification. We needed to perform additions to the theoretical system, to assure the correct extraction of shared communications, more precisely we introduced the wildcard

polarity, which was not defined previously in the theoretical system in which our work is based.

In addition to the command line tool, we also developed an eclipse plugin which allows the user to use our tool in an eclipse environment.

As future work the tool will be extended so as to support multi-party interaction, following the approach of conversation types introduced by Caires and Vieira [1], and the possibility of automatically infer the event identifiers through an algorithm that identifies the events and the partial order from the code, giving the user the possibility to just specify the (classic) session types. Possible future work also includes the development of a intermediate language to serve as "bridge" between the type checking procedures demonstrated in this work and a more practical environment of development. Such language would give the developer an opportunity to statically verify small critical zones of code that use point-to-point communication in a high-level language, allowing to identify local deadlocks prior to the code execution.

Bibliography

[1] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.

[2] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[3] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[4] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[5] Robin Milner. *Communicating and mobile systems: the Pi-calculus*. Cambridge University Press, 1999.

[6] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, 1992.

[7] Luca Padovani. From lock freedom to progress using session types. In *6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, EPTCS 137, pages 3–19, 2013.

[8] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[9] Hugo Torres Vieira and Vasco Thudichum Vasconcelos. Typing progress in communication-centred systems. In *15th International Conference on Coordination Models and Languages*, volume 7890 of *LNCS*, pages 236–250. Springer, 2013.